Université de Liège

Ingénieur Civil - Master in computer science

INFO0085-1

Compilers

The VSOP compiler

EL FENICHE Kamal - 20171282
HERNANDEZ CAPEL Esteban - 20174817
AMAR Salah - 20170980

15 mai 2021

1 Introduction

The goal of this project is to implement a compiler for the VSOP language. To do this, we used the `C++` language, `flex` and `bison`.

The compiler is divided into four parts : a lexer that performs the lexical analysis, a parser to perform the syntax analysis, a semantic tool to perform the semantic analysis, and a `LLVM` tool to perform code generation and to generate the executable. Each tool works with the previous one. This means that when a step finishes, and no error occurs, the following step is performed.

2 Lexical analysis

The lexical analysis part was done thanks to `flex`.

In order to perform the lexical analysis, one needs to create regular expressions to recognise the words belonging to the language.

Then, in order to handle nested comments, a stack was used so that when a (* is unmatched, we can recover its position. The same protocol was used to handle unmatched opening " for strings. In fact, we defined *starting states* for each of these cases. When we encounter whether a (* or a ", we enter into a state and in this state we collect the information about the string, or nothing if we are in a comment. This allow us to check if we reach the end of file before these symbols are correctly matched.

Moreover, the keywords of the VSOP language do not have specific rules in the lexer. Rather, when the lexer is recognising an *object-identifier* we check if this *object-identifier* is a keyword of the language. If yes, the corresponding token is returned. This helps to have a smaller and faster finite deterministic automaton.

If an element does not match any rules, an error message is reported and the execution is stopped. There is no error recovery.

3 Syntax analysis

The syntax analysis was done thanks to `Bison`. It is a bottom-up LALR parser and it works in a bottom-up way. The fact that this parser works in a bottom-up fashion is useful for us to construct the Abstract Syntax Tree (AST).

The parser works as follow : it uses the lexer to generate the token one by one. And for each token, it tries to match a rule. For each matched rule, it executed the corresponding instructions which create a new node inside the AST.

An object-oriented inheritance based approach was chosen for this part of the project. Each node of the AST is an object and contains references to its child nodes. The root node of the AST is an object called `VSOPProgram` which contains a reference to the list of `Class` objects that represents the classes defined in the VSOP file. Then, each `Class` object will have a reference to a list of `Method` and a list of `Field` which represents its methods and fields. And each `Method` object will have references to its `Formal`, and so on...

In order to gain advantage of the inheritance, each class in the AST inherits from a `Node` class, and each expression inherit from a `Expr` class (which itself inherits from `Node`).

When an error occurs during the parsing, the error is reported to the user. We did not implement any kind of error recovery. Therefore, in our implementation, once there is an error during the parsing, it prints on the standard error and stops. However, we managed for some case to print some specific error message. Indeed, when the parser is expecting an *object-identifier* but receives a *type-identifier* we print a special message, and vice-versa.

4 Semantic analysis

For the semantic analysis no special tool was used, we implemented it ourselves. We used our object-oriented approach to implement it.

We have defined three passes in our AST in order to check the types of each node.

The first pass consists of declaring each class including the `Object` class, and checking if classes are not redefined. Also, for each class we check if does not redefine its own fields and methods. For the methods, we check if it does not redefine its formals. After everything was declared, we check if the main class is present alongside with the main function with the correct signature.

The second pass consists of checking the inheritance, *i.e.* checking if there is no cycle in the inheritance. In order to detect cycles, we iterate on the ancestor of this class and check if the ancestor name is equal to this class. If yes, then it means that this class is having a cycle in its inheritance.

The last and third pass consists of checking for each class if it does not override inherited field and methods.

In order to keep track of the scope, we implemented symbol tables. We have implemented a `SymbolTable` class which represents a symbol table (!). In order to keep track of the scope and of the defined variables, each element that introduces a new scope add its own fields, *i.e.* a class would push its fields and a method would push its formals, a `let` would push its name. In order to resolve the fact that a formal of a method can overshadow a field of a class (if they share the same name), the symbol table is in fact a mapping between a name and a vector of type. So that popping a name from the symbol table removes the last element of the vector corresponding to this name. And pushing correspond to add a new entry at the end of the vector.

In this phase, the error reporting was made by ourselves and we could report as many error messages as we wanted. When we encounter an undefined type or identifier, we return an `unknown` type.

5 Code generation

In order to generate code, the `LLVM C++` API was used. Before generating code, we had to make another traversal in the AST to declare classes and methods inside the `LLVM` *module*. For the classes we had to define its Vtable, its fields and methods. But also the also the `new` and `init` function of each class. Also, a `CodeGenerator` class was implemented it contains a *builder*, a *context*, and a *module*. These three elements are used very often. So to ease their usage, they were joined in this class. Also, this class implements a kind of symbol table. It works the same as previously but this time it maps a name to a vector of `llvm::Value`*, which represents a SSA value.

Thanks to the `LLVM C++` API several optimization passes were done. For example, we used an optimization pass to remove unreachable blocks.

The `unit` type was interpreted like the `void` type. So, in order to handle elements that have this type we just push a `nullptr` into the symbol table. Therefore, a `unit` type does not require any *store* or *allocate* method.

It is good to notice that we supposed that no error could happen during this phase. So, we have implemented this phase without some verification mechanism (for example, checking if a type exists or not, ...).

6 Extensions

No extension was implemented.

7 Limitations

One of the limitation of our compiler is about error handling. Indeed, for the first two parts of the compiler (lexical and syntax analysis) we do not handle the errors very well. In the lexical analysis, when an undefined element

is encounter an error message is reported and that's it. There is no error recovery, such as printing a default token. For the syntax analysis, the message error are just the verbose `Bison` error messages. The only defined message error we print is when a *type-identifier* is used instead of a *object-identifier*, and vice-versa. And, the parser stops immediately after one error.

8 Retrospective analysis

We are quite satisfied by the final result as it works on all the tests on the submission plateform. This project has been a *hard* and *challenging* project which helped us a lot to understand the theoretical course.

Also, we could not implement some extensions due to some lack of time but our main goal was first to produce a correct compiler.

Finally, we spend a lot of time in the last part of the project : code generation. Indeed, we choose to use the `LLVM C++` API and this library is quite undocumented even though the tutorial was very helpful to understand how it works, but it was hard to understand it.

9 Conclusion

This project helped us a lot to understand how compilers works and how hard it is to implement one.

We spent a large amount of time in this project, and we estimate that each person in the group has spent 70 hours in this project.